

AFRL-IF-RS-TR-2003-52
Final Technical Report
March 2003



ENGINEERING A DISTRIBUTED INTRUSION TOLERANT DATABASE SYSTEM

University of Maryland Baltimore County

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K445

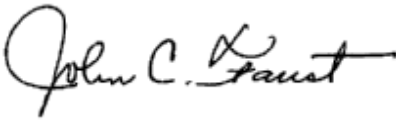
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.


The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-52 has been reviewed and is approved for publication.

APPROVED: 
JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: 
WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE MARCH 2003	3. REPORT TYPE AND DATES COVERED Final May 00 – May 02	
4. TITLE AND SUBTITLE ENGINEERING A DISTRIBUTED INTRUSION TOLERANT DATABASE SYSTEM			5. FUNDING NUMBERS C - F30602-00-2-0575 PE - 62301E/63760E PR - K445 TA - 15 WU - A1	
6. AUTHOR(S) Peng Liu				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Baltimore County 1000 Hilltop Circle Baltimore Maryland 21250			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-52	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: John C. Faust/IFGB/(315) 330-4544/John.Faust@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The primary accomplishment of this project is a new paradigm for secure database system design, intrusion tolerant database systems. In particular, an innovative intrusion tolerant database system framework, denoted ITDB, is developed. While traditional secure database systems rely on preventive controls, ITDB can detect intrusions, isolate attacks, contain, assess and repair the damage caused by intrusions in a timely manner such that a self-stabilized level of data integrity and availability can be provided to applications. Built on top of COTS DBMS, ITDB arms commercial database servers with the ability to deliver sustained valid data access services even in the face of intensive attacks. To validate ITDB, a prototype ITDB system is designed and implemented. The prototype is a seamless integration of five major subsystems, namely the Malicious Transaction Detection subsystem, the Attack Recovery subsystem, the Attack Isolation subsystem, the Damage Containment subsystem, and the Self-Stabilization subsystem. Extensive evaluation of the prototype based on practical database applications, simulated workload and injected attacks is done. Preliminary testing measurements suggest that when the accuracy of the intrusion detector is satisfactory, ITDB can effectively tolerate database intrusions with reasonable performance penalty.				
14. SUBJECT TERMS Intrusion Tolerance, Intrusion Tolerant Database Systems, Information Assurance, Survivability			15. NUMBER OF PAGES 28	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. INTRODUCTION.....	1
1.1 Background and Prior Work	1
1.2 ITDB Framework.....	3
1.3 Attack Prediction	5
2. MALICIOUS TRANSACTION DETECTION SUBSYSTEM.....	6
2.1 The Goal of ITDB	6
2.2 Application aware database intrusion detection	7
3. ATTACK RECOVERY SUBSYSTEM.....	9
4. ATTACK ISOLATION SUBSYSTEM.....	11
5. DAMAGE CONTAINMENT SUBSYSTEM.....	14
6. SELF-STABILIZATION SUBSYSTEM	17
7. ITDB PROTOTYPE	20
8. ITDB VALIDATION FRAMEWORK	20
9. ATTACK PREDICTION	21
10. REFERENCES.....	23

List of Figures

Figure 1 - ITDB Architecture	4
Figure 2 – ITDB Multilayer Methodology	5
Figure 3 – Architecture I	7
Figure 4 – ITDB Intrusion Detection Subsystem	8
Figure 5 – Organization of the Attack Recovery Subsystem Prototype	11
Figure 6 – Architecture II	12
Figure 7 – Organization of the Attack Isolation Subsystem Prototype.....	14
Figure 8 – Architecture III.....	15
Figure 9 – Organization of the Damage Containment Subsystem Prototype	17
Figure 10 – Architecture IV.....	18
Figure 11 – Organization of the Self-Stabilization Subsystem Prototype	19
Figure 12 – ITDB Prototype Installation	21
Figure 13 – Model of Game Theoretic Attack Prediction	22

List of Tables

Table 1 – Vulnerabilities of Existing Database Assurance Mechanisms.....	2
--	---

1. Introduction

The visions of Internet applications (e.g., e-commerce) and pervasive computing not only push computations from a computer into everywhere, but also maximize our dependence on networked computing systems. Quickly increased complexity, openness, inter-connection, and inter-dependence have made these systems more vulnerable and difficult to protect than ever. The inability of existing security mechanisms to prevent every attack is well embodied in several recent large scale Internet attacks such as the DDoS attack in Feb. 2000 [Taylor00]. These accidents convince the security community that traditional *prevention-centric* security is not enough and the need for *intrusion tolerant* or *attack resilient* systems is urgent. Intrusion tolerant systems, with characteristics quite different from traditional secure systems, extend traditional secure systems to *survive* or *operate through* attacks. The focus of intrusion tolerant systems is the ability to continue delivering essential services in the face of attacks.

The primary goal of this project is to develop ITDB, an Intrusion Tolerant Database System framework, and prototype ITDB systems.

The ITDB project has two major accomplishments: (1) the ITDB framework and (2) a (preliminary) game-theoretic attack prediction model. The ITDB framework is the focus of this project.

1.1 Background and Prior Work

Being a critical component of almost every mission critical information system, database products are today a multi-billion dollar industry. Database systems motivated 32% of the hardware server volume in 1995 and 39% of the server volume in 2000. Improving the intrusion tolerance of database systems has a direct positive impact on the technology that enables a variety of critical, trusted applications such as e-commerce, air traffic control, credit-card, telecommunication control, and electricity and water supply systems, that our everyday life depends on.

However, existing database security mechanisms are very limited in *tolerating* or *surviving* intrusions. In particular, authentication and access control cannot prevent all attacks; integrity constraints are weak at prohibiting plausible but incorrect data; concurrency control and recovery mechanisms cannot distinguish legitimate transactions from malicious ones; and automatic replication facilities and active database triggers can even serve to spread the damage.

As a result, although current commercial “off-the-shelf” (COTS) database management systems (DBMS) are equipped with pretty good preventive mechanisms such as authentication and access control, information assurance of existing database applications built on top of these COTS DBMS is seriously threatened by the lack of (good) survivability, which can cause data to be damaged (without being detected), wrong decisions to be made based on damaged data, the data integrity level to be seriously decreased, and the availability of the database to be (indirectly) jeopardized.

The vulnerabilities of existing database assurance mechanisms in tolerating intrusions can be summarized by Table 1.

Table 1 – Vulnerabilities of Existing Database Assurance Mechanisms

VA1	Access controls can be subverted by the inside attacker, or the outside attacker who assumes the insider's identity.
VA2	Integrity constraints are weak at prohibiting plausible but incorrect data.
VA3	Concurrency-control and recovery mechanisms cannot distinguish an attacker's transactions from any other legitimate transaction.
VA4	OS and lower level data corruption attacks can corrupt the database. Corrupted data can lead to wrong (real world) decisions or actions, which can be dangerous, harmful, misleading, and disaster-prone.
VA5	Malicious transactions can seriously corrupt the data. Corrupted data can lead to wrong (real world) decisions or actions, which can be dangerous, harmful, misleading, and disaster-prone.
VA6	Data corruption caused by malicious transactions (and lower level attacks) can force the database server to halt periodically to assess and repair the damage, which hurts the availability.

Making a database system *intrusion tolerant* is, in general, a *multi-layer* job, since the attacks could come from any of the following layers: hardware, OS, DBMS, and transactions (or applications). The multi-layer approach is being developed along two directions: (1) from scratch or (2) using COTS components.

Along the from-scratch direction, tamper resistant processing environments, and trusted OS or trusted DBMS loaders have been applied to close the door for hardware attacks and OS bugs; trusted DBMS have been applied to close the door for DBMS bugs; and signed checksums (and a small amount of tamper resistant storage to keep the signing key) are used to detect OS level data corruptions [MVS00]. However, the from-scratch approach is usually not a cost-effective approach, and it cannot be used to tolerate authorized transaction level intrusions, especially VA5 and VA6.

Based on COTS components, OS level attacks are addressed by several efforts. In [BGI00], (signed) checksums are smartly used to detect data corruption. In [MG96], a technique is proposed to detect *storage jamming*, malicious modification of data, using a set of special *detect objects* which are indistinguishable from normal objects by the jammer. Modification of detect objects indicates a storage jamming attack. Although these can be used to effectively tolerate OS level intrusions, they cannot handle authorized but malicious transactions, especially VA5 and VA6.

In summary, although existing database survivability techniques can achieve pretty good resilience to OS (and other lower) level attacks, none of them can handle application level or transaction level attacks, namely *malicious* transactions, which represent most of the existing database attacks. The goal of the ITDB framework is to fill this hole. In addition, data corruption

directly caused by lower level attacks can spread across the database through the read and write operations of (innocent) transactions. OS-level survivability techniques cannot handle the damage spreading, but the ITDB framework can.

The ITDB framework can be compared with three types of database systems: (1) COTS DBMS such as Oracle, SQL Server, Sybase, and Informix; (2) trusted database systems such as ITB [MVS00]; (3) COTS DBMS enhanced with OS-level survivability tools such as data jamming [MG96] and data corruption detection [BGJ00]. In terms of integrity, type (1) systems use integrity constraints to protect the database from inconsistent data. However, they cannot prevent the attacker from corrupting data without violating data consistency. Type (2) and Type (3) systems can effectively detect OS-level data corruption. However, none of the three types of systems can handle transaction-level data corruption or damage spreading. These kinds of integrity threats can only be addressed by ITDB. In terms of availability, these three types of systems may need to halt the database during the repair. However, ITDB systems inherently support WarmStart repair where the database server never stops. Finally, it should be noticed that ITDB is built on top of existing database security and survivability techniques. An ITDB system can be built on top of either a COTS DBMS, or a trusted database system. Therefore, the confidentiality, authentication, and non-repudiation of an ITDB system rely on the corresponding information assurance attributes of the underlying system.

1.2 ITDB Framework

The ITDB framework focuses on data integrity and availability. ITDB does not address confidentiality, authentication, or non-repudiation. Within the ITDB framework, at one point of time, data *availability* is defined by the set of data objects that are *available* (or *accessible*), and the *data availability level* is (roughly) measured by the percentage of the data objects that are available. Data *integrity* is defined by the set of data objects that are corrupted, whether these data objects are available or not, and the *data integrity level* is (roughly) measured by the percentage of the data objects that are corrupted. Note that at one point of time two database servers can have the same data integrity (availability) level, but very different data availability (integrity) levels.

The goal of ITDB is to use COTS components to build database servers that can maintain not only a desired level of data integrity, but also a desired level of data availability in the face of attacks. In this way, database servers can have significantly improved ability to deliver sustained correct (or valid) data access (or transaction processing) services even in face of intensive attacks.

An ITDB system, with the architecture shown in Figure 1, can detect malicious transactions, isolate malicious transactions, contain, assess, and repair the damage caused by malicious transactions and other lower level attacks in such a way that a self-stabilized level of data integrity could be provided to applications [LJLI01].

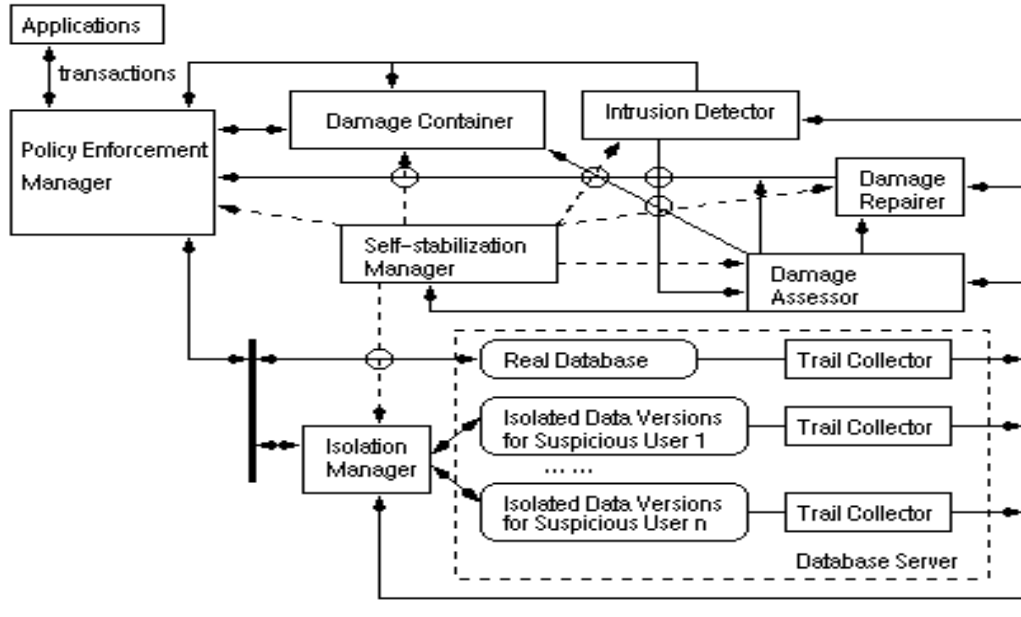


Figure 1 - ITDB Architecture

An ITDB system is a *concurrent* system triggered by a set of specific *events*. The major events are: (1) when a *user* transaction T is submitted by an application, the PEM (*Policy Enforcement Manager*) will proxy each SQL statement and transaction processing *call* of T , and keep useful information about T and these SQL statements; (2) when an SQL statement is executed, the *Trail Collectors* will log the corresponding writes in the *Write Log*, the corresponding reads will instead be extracted from the statement text into the *Read Log*, and the *Intrusion Detector* will assess the *suspicion level* of the corresponding transaction and session using the trails kept in the *Write Log* (and possibly some other audit tables); (3) when the *Intrusion Detector* identifies a *suspicious* user, the PEM will notify the *Isolation Manager* to start isolating the user; (4) when the *Intrusion Detector* identifies a *malicious* transaction, the *Damage Assessor* will start to locate the damage caused by the transaction, and the PEM and the *Damage Container* will start the multi-phase damage containment process (if needed); (5) when the damage caused by a malicious or affected transaction is located, the *Damage Repairer* will compose and submit a specific *cleaning* transaction to repair the damage; (6) when the *Damage Assessor* finds an unaffected transaction, when the *Damage Container* identifies an undamaged object, or when a cleaning transaction commits, some objects will be reported to the PEM to do uncontainment; (7) when the *Intrusion Detector* finds that a suspicious user is malicious, the *Isolation Manager* will discard the isolated data versions maintained for the user; (8) when the *Intrusion Detector* finds that a suspicious user is actually innocent, the isolation manager will merge the work of the user back into the real database by composing and submitting some specific *back-out* and/or *update-forwarding* transactions (to the PEM); (9) when the *Self-Stabilization Manager* receives a report from some other components such as the *Damage Assessor*, some *reconfiguration* commands could be generated and sent to some other ITDB components.

From the perspective of defense-in-depth survivability, ITDB’s methodology can be summarized by Figure 2, where (a) there are multiple layers (or phases) of intrusion tolerance operations and (b) lower layer operations usually build the foundations for higher layer operations, although in some cases operations at several layers could be done concurrently. Compared with the methodology of making the resilience of a system dependent on only one or two mechanisms such as intrusion detection, ITDB’s methodology is not only comprehensive, but also more resilient to attacks.

6	Damage Repair		Reconfiguration	
5	Damage Containment			
4	Damage Assessment			
3	Merging	-----		
	Isolation			
2	Intrusion Detection			
1	Access Control			

Figure 2 – ITDB Multilayer Methodology

The ITDB architecture assures integrity by dynamically maintaining a self-stabilized level of data integrity (with the cost of some availability loss). The damage caused by a malicious transaction will be detected and identified through intrusion detection and damage assessment. A substantial amount of damage will be isolated without causing any harm to the (main) database. Located damage will be repaired on-the-fly. A cost-based self-tuner is used to stabilize the data integrity level through agile adaptive reconfiguration. The ITDB architecture assures availability by WarmStart damage assessment and repair (without halting the database), damage containment (without denying the access to undamaged parts of the database), and attack isolation (without rejecting suspicious transactions). The availability loss caused by flooding attacks to databases is not addressed by ITDB.

1.3 Attack Prediction

The ability to *predict* (the actions of) attacks can significantly enhance people’s ability to build intrusion tolerance systems due to a couple of reasons. First, one very desired feature of an intrusion tolerant system is that it can deliver *quantitative* information assurance guarantees, that is, its resilience can be *measured*. One of the key reasons that existing intrusion tolerant systems do not have this feature and existing security evaluation techniques cannot measure information assurance is that the resilience of an intrusion tolerant system is heavily dependent on the attacks; however, attacks are *intentionally* setup and very difficult to predict. Hence the ability to model and predict attacks is a critical step towards measurable information assurance. Second, the ability to predict attacks has the potential to transform existing *passive* (or reactive) secure systems, where the defender lags behind the attacker, into *active* ones.

Attack prediction can be broken down into two categories: trend prediction and action prediction. In this project, a (preliminary) game theoretic approach is developed for attack prediction, which

is the first approach for action prediction (based on the relevant literature review). This approach models the computer system and the attacker(s) as two *self-interested players* playing a multi-stage *game* where the system wants to maximize its security through its defense operations while the attacker wants to maximize the security loss through his or her attacks. The *Nash equilibria* of the game, which specify the expected-utility maximizing best-response of one player to every other player, indicate valuable action *predictions*. In addition to predicting attacks, the predictions generated by our approach can also give a good estimation of the maximum possible security loss and tell how the defense should be built. It is believed that this approach can be used to predict almost every known type of attacks. In particular, a general game-theoretic attack prediction model for attacks on IDS-protected systems is presented, and a specific prediction model for credit card fraud is presented, and the preliminary simulation results are very encouraging. In Section 9, more details about this accomplishment will be presented.

2. Malicious Transaction Detection Subsystem

From Section 2 to Section 6, the key components of the ITDB framework will be presented. For clarity, the ITDB framework is broken down into four evolving schemes where every later-on scheme is built on top of the previous schemes.

2.1 The Goal of ITDB

Since the property of database *atomicity* indicates that only committed transactions can really change the database, it is theoretically true that if every malicious transaction can be detected before it commits, then the transaction can be rolled back before it causes any damage. However, this “perfect” solution is not practical for two reasons. First, transaction execution is, in general, much quicker than detection, and slowing down transaction execution can cause very serious denial-of-service. For example, the Microsoft SQL Server can execute over 1000 (TPC-C) transactions within one second (see www.oracle.com), while the average anomaly detection latency is typically in the scale of minutes or seconds. Detection is much slower since: (1) in many cases detection needs human intervention; and (2) to reduce false alarms, in many cases a sequence of actions should be analyzed.

Second, some authorized but malicious transactions are very difficult to detect. They look and behave just like other legitimate transactions. Anomaly detection based on the semantics of transactions (and the application) may be the only effective way to identify such attacks; however, it is very difficult, if not impossible, for an anomaly detector to have 100% detection rate with reasonable false alarm rate and detection latency.

Hence, a *practical* goal should be: “after the database is damaged, locate the damaged part and repair it as soon as possible, so that the database can continue being useful in face of attacks.” In other words, the database system is designed to operate through attacks.

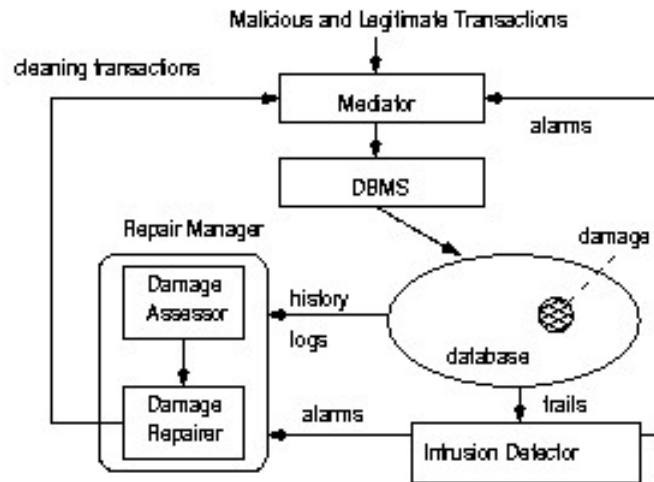


Figure 3 – Architecture I

Architecture I, which is shown in Figure 3, combines intrusion detection and attack recovery to achieve this goal. In particular, the *Intrusion Detector* monitors and analyzes the *trails* of database sessions and transactions in a real-time manner to identify malicious transactions as soon as possible. Alarms of malicious transactions, when raised, will be instantly sent to the *Repair Manager*, which will locate the damage caused by the attack and repair the damage. During the whole intrusion detection and attack recovery process, the database continues executing new transactions.

2.2 Application aware database intrusion detection

Although there are a lot of anomaly detection algorithms (for host or network based intrusion detection) [Lunt93, MHL94], they usually cannot be directly applied in malicious transaction detection, which faces the following unique challenges:

- Application semantics must be captured and used. For example, for a school salary management application, a \$3000 raise is normal, but a \$10000 raise is very *abnormal*. Application semantics based intrusion detection is *application aware*. Since different applications can have very different semantics, general application-aware database intrusion detection systems must support dynamic integration of application semantics. Since different anomaly detection algorithms may be good for different application semantics, a general application-aware database intrusion detection system must adapt itself to application semantics.
- Multi-layer intrusion detection is usually necessary for detection accuracy. First, proofs from application layer, session layer, transaction layer, process layer, and system call layer should be *synthesized* to do intrusion detection. Lower level proofs can help identify higher level anomalies. Second, OS-level and transaction-level intrusion detection should be coupled with each other.

Within the ITDB project, a *cartridge* like detector is designed to address these challenges. The detector is a cartridge which is general enough to plug in a variety of (a) anomaly detection algorithms called *bullets*, (b) *application semantics* extraction algorithms, and (c) application semantics based *adaptation policies*. The user is able to prepare some of these algorithms and policies. The detector provides the interfaces for the user to pick existing and provide new bullets, and the detector is not required to rebuild itself again and again to support each new bullet. In this way, the detector can be used to meet the intrusion detection needs of multiple applications. Flexibility and expressiveness are the major merits of this detector.

A simple cartridge like detector is implemented within the ITDB project where bullets are supported through DLL (Dynamic Linkable Libraries) modules and a rule-based mechanism is used to build the cartridge. The architecture of the detector is shown in Figure 4. In general, rules are used for two purposes: (1) application semantics are programmed as *rules*, and (2) bullets are plugged in as one or more rules. The rules are stored in the *Rule Base*. An *interface* (though not shown in the figure) is provided for the security officer to dynamically register rules and manage the Rule Base. A rule is fired and processed by the *Rule Processor* when a specific *event* is generated by the *Event Generator*. The Event Generator generates events based on the trails collected by the *Mediator* and some other Trail Collectors such as the set of ITDB triggers. All the activities are coordinated by the *Intrusion Monitor*, which is responsible for raising the *suspicion levels*. At this stage, ITDB has implemented two bullets: one is a rule-based anomaly detection algorithm; the other is a data mining based anomaly detection algorithm. Readers can refer to [LIM01] for more details about the intrusion detection subsystem.

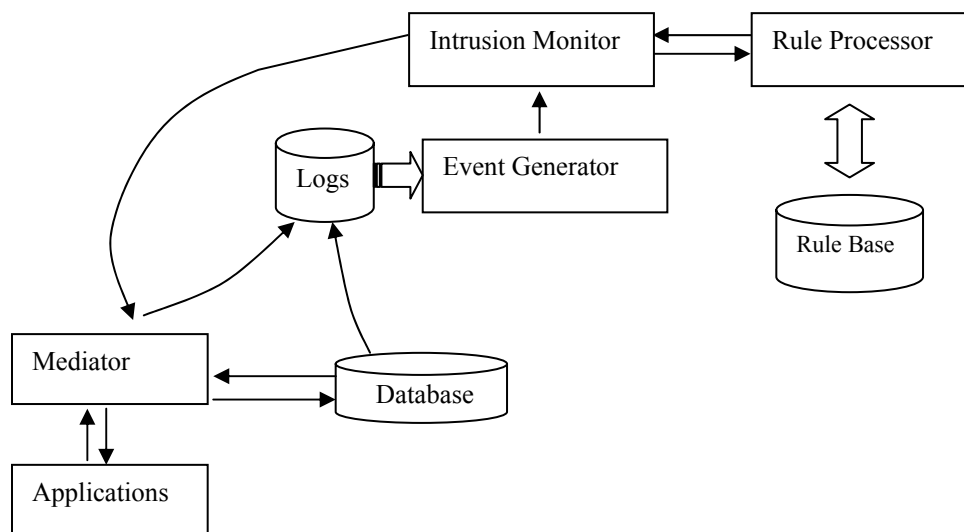


Figure 4 – ITDB Intrusion Detection Subsystem

3. Attack Recovery Subsystem

Malicious transactions can seriously corrupt a database through a vulnerability denoted *damage spreading*. In a database, the results of one transaction can affect the execution of some other transactions. When a transaction T_i reads a data object x updated by another transaction T_j , T_i is directly *affected* by T_j . If a third transaction T_k is affected by T_i , but not directly affected by T_j , T_k is indirectly affected by T_j . It is easy to see that when a (relatively old) transaction B_i that updates x is identified malicious, the damage on x can spread to every object updated by a *good* transaction that is affected by B_i , directly or indirectly. In a word, the read-from dependency among transactions forms the *traces* along which damage spreads.

The job of attack recovery is two-fold: damage assessment and repair. In particular, the job of the *Damage Assessor* is to locate each affected good transaction, i.e., the damage spreading traces; and the job of the *Damage Repairer* is to recover the database from the damage caused on the objects updated along the traces. In particular, when an affected transaction T is located, the Damage Repairer builds a specific *cleaning* transaction to *clean* each object updated by T (and not cleaned yet). Cleaning an object is simply done by restoring the value of the object to its latest undamaged version.

Temporarily stopping the database will certainly make the attack recovery job simpler since the damage will no longer spread and the repair can be done backwardly after the assessment is done, that is, the database can be repaired by simply undoing the malicious as well as affected transactions in the reverse order of their commit order. However, since many critical database servers need to be 24*7 available and temporarily making the database shut down can be the real goal of the attacker, on-the-fly attack recovery which never stops the database is necessary in many cases.

On-the-fly attack recovery faces several unique challenges. First, ITDB needs to do repair forwardly since the assessment process may never stop. Second, cleaned data objects could be re-damaged during attack recovery. Finally, the attack recovery process may never *terminate*. Since as the damaged objects are identified and cleaned new transactions can spread damage if they read a damaged but still unidentified object, so ITDB faces two critical questions: (1) Will the attack recovery process terminate? (2) If the attack recovery process terminates, can ITDB detect the termination?

To tackle challenge 1, ITDB must ensure that a later on cleaning transaction will not accidentally damage an object cleaned by a previous cleaning transaction. To tackle challenge 2, ITDB must not mistake a cleaned object as damaged, and ITDB must not mistake a re-damaged object as already cleaned. To tackle challenge 3, the PI's previous study in [AJL02] shows that when the damage spreading speed is quicker than the repair speed, the repair may never terminate. Otherwise, the repair process will terminate, and under the following three conditions ITDB can ensure that the repair terminates: (1) every malicious transaction is cleaned; (2) every identified damaged object is cleaned; and (3) further (assessment) scans will not identify any new damage (if no new attack comes).

From a state-transition angle, the job of attack recovery is to get a *state* of the database, which is determined by the values of the data objects, where (a) no effects of the malicious transactions

are there and (b) the work of good transactions should be kept as much as possible. In particular, transactions transform the database from one state to another. Good transactions transform a good database state to another good state, but malicious transactions can transform a good state to a damaged one. Moreover, both malicious and affected (good) transactions can make an already damaged state even worse. A database state S_1 is said *better* than another one S_2 if S_1 has less number of objects corrupted. The goal of on-the-fly attack recovery is to get the state better and better, although during the repair process new attacks and damage spreading could (temporarily) make the state even worse.

Architecture I has the following properties: (1) it builds itself on top of a COTS DBMS. It does not require the DBMS kernel be changed. It has almost no impact on the performance of the database server except that the *Mediator* can cause some service delay and the cleaning transactions can make the server busier. (2) The intrusion tolerance processes are all on-the-fly. (3) During attack recovery, the data integrity level can vary from time to time. When the attacks are intense, damage spreading can be very serious, and the integrity level can be dramatically lowered. In this situation, asking the *Mediator* to slow down the execution of new transactions can help *stabilize* the data integrity level, although this can cause some availability loss. This indicates that integrity and availability can be two conflicting goals in intrusion tolerance. (4) More availability loss can be caused when (a) the Intrusion Detector raises false alarms; or (b) a corrupted object is located (It will not be accessible until it is cleaned. Making damaged parts of the database available to new transactions can seriously spread the damage). (5) Inaccuracy of the Intrusion Detector can cause some damage not located or repaired. (6) Architecture I is not designed to and cannot handle physical world attack recovery, which usually requires many additional activities. Logically repairing a database does not always indicate that the corresponding physical world damage can be recovered.

To justify the cost-effectiveness of Architecture I, a prototype of Architecture I is implemented on top of an Oracle database server (within the ITDB project). The prototype subsystem is shown in Figure 5. In general, the *Triggers* and the *Mediator* log the raw trails of transactions. The *Write Log Generator* uses the raw trails to produce the *Write Log* where the write operations of transactions are kept. The *Read Log Generator* extracts read operations from the raw trails using the read set *templates* extracted from transaction *profiles*. When a malicious transaction is identified by the *Intrusion Detector*, the *Repair Manager* will do both damage assessment and repair. The corresponding cleaning (or undo) transactions will be submitted to the *Mediator* for execution.

The cost-effectiveness of the attack recovery subsystem prototype is evaluated using simulated *workload* and injected attacks. The preliminary testing measurements suggest that when the accuracy of the Intrusion Detector is satisfactory, the prototype can effectively locate and repair the damage on-the-fly with reasonable (database) performance penalty. Readers can refer to [LL01] for more details about this subsystem.

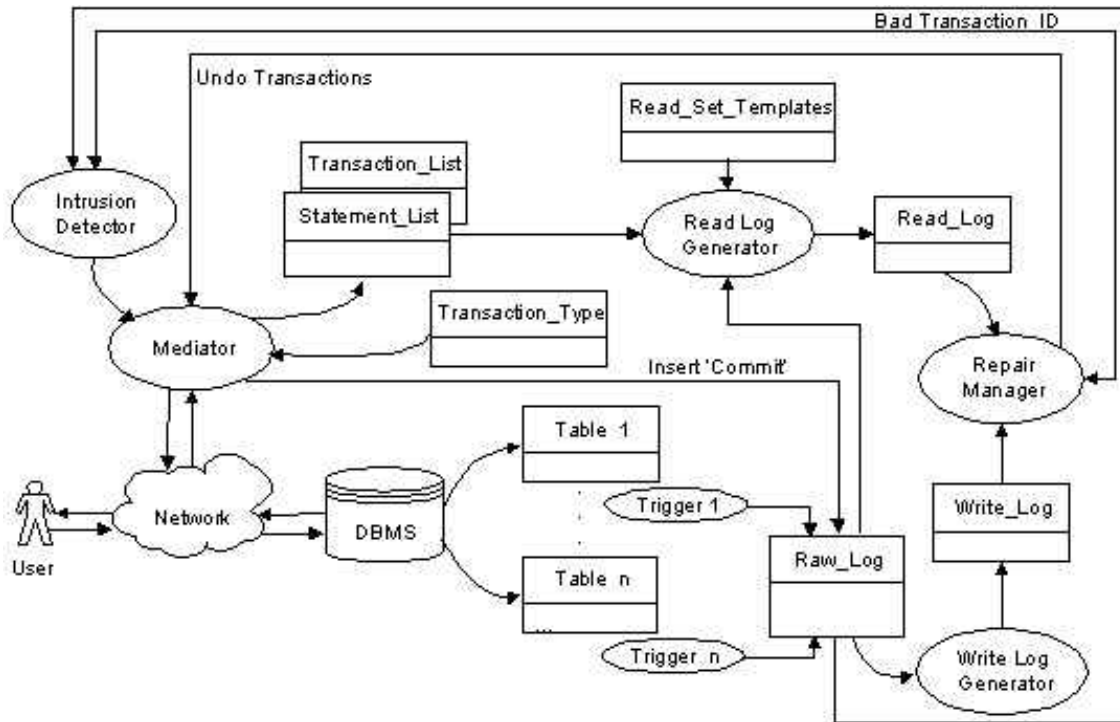


Figure 5 – Organization of the Attack Recovery Subsystem Prototype

Finally, it should be noticed that the Attack Recovery Subsystem can easily handle the damage spread out from the data objects corrupted by OS (and lower) level database attacks.

4. Attack Isolation Subsystem

One problem of Architecture I is that during the *detection latency* of a malicious transaction B, i.e., the duration from the time B commits to the time B is detected, damage can seriously spread. The reason is that during the detection latency many innocent transactions could be executed and affected. For example, if the detection latency is 2 seconds, then the Microsoft SQL Server can execute over 2000 transactions during the latency on a single system, and they can access the objects damaged by B freely (since ITDB does not know which objects are damaged by B during the latency).

Quicker intrusion detection can mitigate this problem; however, reducing detection latency without sacrificing the false alarm rate or the detection rate is very difficult, if not impossible. When the detection rate is decreased, more damage is left unrepaired. When the false alarm rate is increased, more denial-of-service will be caused. These two outcomes conflict with the goal of Architecture I.

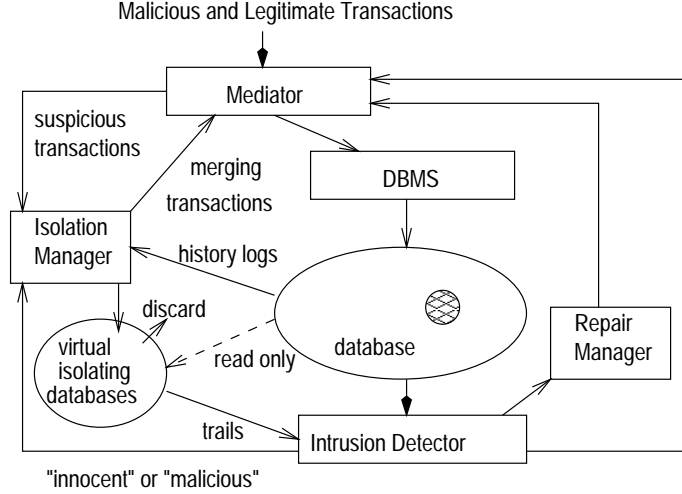


Figure 6 – Architecture II

Architecture II, as shown in Figure 6, integrates a novel isolation technique to tackle this problem. In particular, first, the Intrusion Detector will raise two levels of alarms: when the (synthesized) anomaly of a transaction (or session) is above Level 1 anomaly threshold TH_m , the transaction is reported malicious; when the anomaly is above Level 2 anomaly threshold TH_s (but below TH_m), the transaction is reported *suspicious*. (The values of TH_m and TH_s are determined primarily based on the statistics of previous attacks). Suspicious transactions should have a significant probability to be an attack. Second, when a malicious transaction is reported, the system works in the same way as Architecture I. When a suspicious transaction T_s is reported, the Mediator, with the help of the *Isolation Manager*, will redirect T_s (and the following transactions submitted by the user that submits T_s) to a *virtually* separated database environment where the user will be isolated. Later on, if the user is proven malicious, the Isolation Manager will discard the effects of the user; if the user is shown innocent, the Isolation Manager will *merge* the effects of the user back into the main database. In this way, damage spreading can be dramatically reduced without sacrificing the detection rate or losing the availability of good transactions.

ITDB does isolation user-by-user because the transactions submitted by the same user (during the same session) should be able to see the effects of each other. And the framework should be able to isolate multiple users at the same time. Isolating a group of users within the same virtual database can help tackle collusive attacks; however, a lot of availability can be lost when only some but not all members of the group are malicious. Using a completely replicated database to isolate a user has two drawbacks: (1) it is too expensive; (2) new updates of unisolated users are not visible to isolated users. In Architecture II, ITDB uses data *versions* to virtually build isolating databases. In particular, a data object x always has a unique trustworthy version, denoted $x[main]$. And only if x is updated by an isolated user can x have an extra suspicious version. In this way, the total number of suspicious versions will be much less than the number of main versions.

The isolation algorithm has two key parts: (1) how to perform the read and write operations of isolated users (Note that unisolated users can access only the main database); and (2) how to do

merging after an isolated user is proven innocent. For part 1, ITDB enforces *one-way* isolation where isolated users can read main versions if they do not have the corresponding suspicious versions, and all writes of isolated users must be performed on suspicious versions. In this way, the data freshness to isolated users is maximized without harming the main database.

The key challenge in part 2 is the inconsistency between main versions and suspicious versions. If a trustworthy user and an isolated user update the same object x independently, $x[main]$ and the suspicious version will become inconsistent, and one update has to be backed out in order to do consistent merging. In addition, our (previous) study in [LJM99] shows that (1) even if they do not update the same object, inconsistency could still be caused; and (2) the merging of the effects of one isolated user could make another still being isolated history invalid. These inconsistencies must be resolved during a merging (e.g., [LJM99] proposes a *precedence-graph* based approach that can identify and resolve all the inconsistencies).

Architecture II has the following properties: (1) Isolation is, to a large extent, transparent to suspicious users. (2) The extra storage cost for isolation is extremely low. (3) The data consistency is kept before isolation and after merging. (4) During a merging, if there are some inconsistencies, some isolated or unisolated transactions have to be backed out to resolve these inconsistencies. This is the main cost of Architecture II. Fortunately, the simulation study done in [D84] shows that the back-out cost is only about 5%. After the inconsistencies are resolved, the merging can be easily done by *forwarding* the remaining updates of the isolated user to the main database. (5) Architecture II has almost no impact on the performance of the database server except that during each merging process (a) the isolated user cannot execute new transactions; and (b) the main database tables involved in the update forwarding process will be temporarily locked.

An isolation subsystem prototype, which is shown in Figure 7, is implemented to further justify the cost-effectiveness of Architecture II. In general, the *Intrusion Detector* informs which users are suspicious and should be isolated. The *Mediator*, which has three components, *proxies* every user transaction and SQL statement (or command). The *triggers*, the *SQL Statement Logger*, and the *Read Extractor* are responsible for keeping track of the read and write operations of transactions, which are necessary to build the precedence graph when a merging should be done. The *SQL Statement Rewriter and Redirector* (SRR) is responsible for enforcing Part I of the isolation algorithm. The *Merger* is responsible for enforcing Part II of the isolation algorithm, namely (a) inconsistency identification and resolution and (b) the Merging Algorithm. The *On-the-fly Isolation Controller* enables new user transactions to continue executing without jeopardizing the correctness of merging processes. In order to transparently isolate a transaction on top of a commercial single-version DBMS such as Oracle, ITDB (a) uses extra tables to simulate multiple versions and (b) rewrites the SQL statements involved in the suspicious transactions in such a way that the one-way isolation policy can be achieved. Note that query rewriting could cause some service delay to isolated users but not to unisolated users. Readers can refer to [Liu01] for more details about this subsystem.

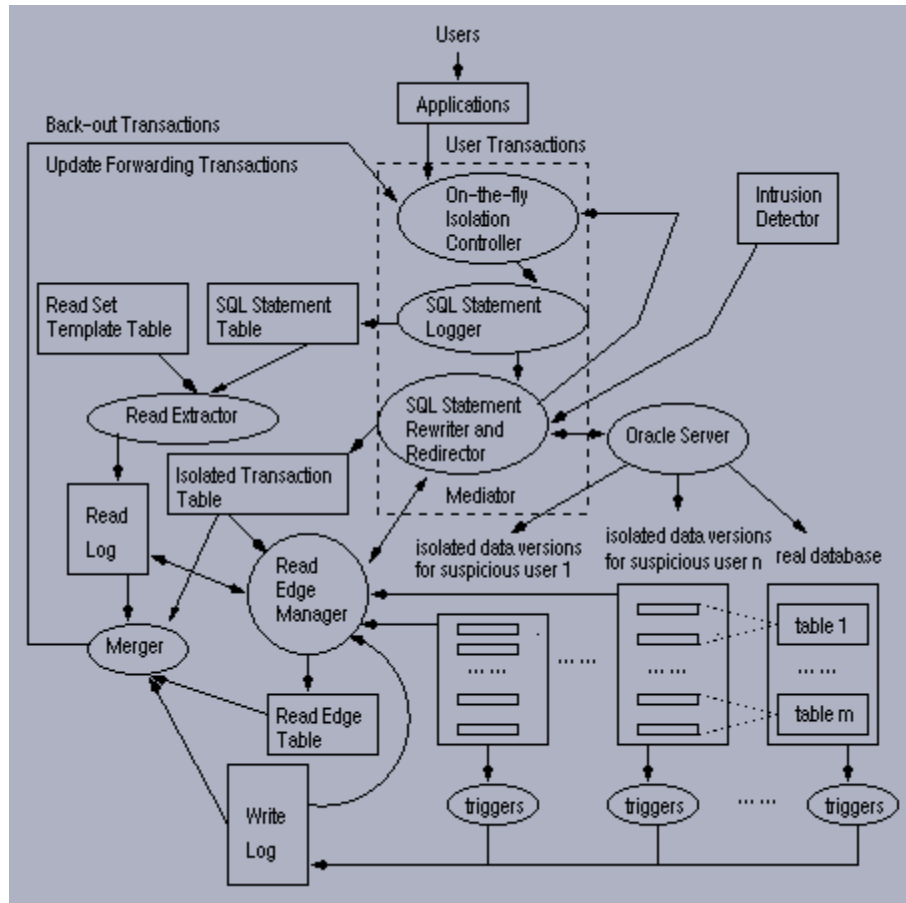


Figure 7 – Organization of the Attack Isolation Subsystem Prototype

5. Damage Containment Subsystem

Another problem of Architecture I is that its damage containment may not be effective. Architecture I *contains* the damage by disallowing transactions to read the set of data objects that are identified (by the Damage Assessor) as corrupted. This *one-phase* damage containment approach has a serious drawback; that is, it cannot prevent the damage caused on the objects that are corrupted but not yet located from spreading. Assessing the damage caused by a malicious transaction B can take a substantial amount of time, especially when there are a lot of transactions executed during the detection latency of B. During the *assessment latency*, the damage caused during the detection latency can spread to many other objects before being contained.

Architecture III, as shown in Figure 8, integrates a novel multi-phase damage containment technique to tackle this problem. In particular, the damage containment process has one containing phase, which instantly contains the damage that *might* have been caused (or spread) by the intrusion as soon as the intrusion is detected; and one or more later-on uncontainment phases to uncontain the objects that are mistakenly contained during the containing phase, and the objects that are cleaned.

In Architecture III, the *Damage Container* will enforce the containing phase (as soon as a malicious transaction is reported) by sending some containing instructions to the *Containment Executor*, and the *Uncontainer*, with the help from the Damage Assessor, will enforce the uncontainment phases by sending some uncontainment instructions to the Containment Executor. The Containment Executor controls the access of the user transactions to the database according to these instructions.

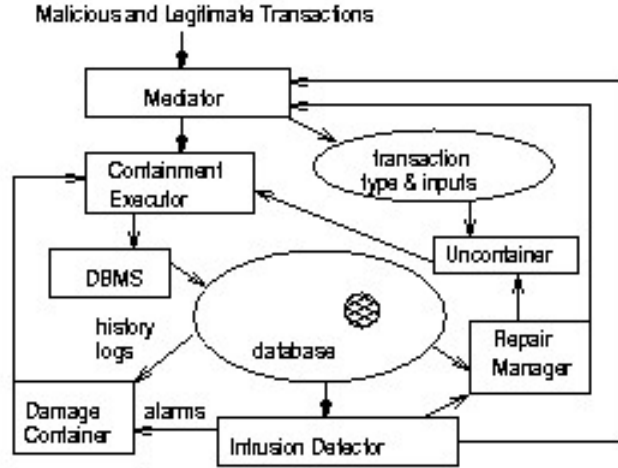


Figure 8 – Architecture III

When a malicious transaction B is detected, the containing phase must ensure that the damage caused directly or indirectly by B will be contained. In addition, the containing phase must be quick enough because otherwise either a lot of damage can leak out during this phase, or substantial availability can be lost. Time stamps can be exploited to achieve the desired goal. The containing phase can be done by just adding an access control rule to the Containment Executor, which denies the access to the set of objects updated during the period of time from the time B commits to the time the containing phase starts. This period of time is called the *containing time window*. When the containing phase starts, every active transaction should be aborted because they could spread damage. New transactions can be executed only after the containing phase ends.

It is clear that the containing phase over-contains the damage in most cases. Many objects updated within the containing time window can be undamaged. And ITDB must uncontain them as soon as possible to reduce the corresponding availability loss. Accurate uncontainment can be done based on the reports from the Damage Assessor, which could be too slow due to the assessment latency. ITDB exploits transaction *types* to do much *quicker* uncontainment. In particular, assuming that (a) each transaction T_i belongs to a transaction type $type(T_i)$ and (b) the *profile* for $type(T_i)$ is known, the *read set template* and *write set template* can be extracted from $type(T_i)$'s profile. The templates specify the kind of objects that transactions of $type(T_i)$ could read or write. As a result, the *approximate* read-from dependency among a history of transactions can be quickly captured by identifying the read-from dependency among the types of these transactions. Moreover, the type-based approach can be made more accurate by *materializing* the templates of transactions using their inputs before analyzing the read-from dependency among

the types. Readers can refer to [LJ01] for more details about our multi-phase damage containment technique.

Architecture III has the following properties: (1) it can ensure that after the containing phase no damage (caused by the malicious transaction) leaks out; (2) as a result, the attack recovery process needs only to repair the damage caused by the transactions that commit during the containing time window, and the termination problem addressed in Architecture I does not exist any longer; and (3) one-phase containment and multi-phase containment are the two extremes of the spectrum of damage containment methods. In particular, one-phase containment has maximum damage leakage (so minimum integrity) but maximum availability, while multi-phase containment has zero damage leakage (so maximum integrity) but minimum availability. In the middle of the spectrum, there could be a variety of approximate damage containment methods that allow some damage leakage.

To justify Architecture III, a damage containment subsystem prototype, which is shown in Figure 9, is implemented. In general, the *Intrusion Detector* informs DDCS which transactions are malicious. The *Transaction Proxy* proxies user transactions for the purpose of keeping track of the status and the SQL statements of transactions. The *triggers* and the *Read Extractor* are responsible for keeping track of the read and write operations of transactions, which are necessary for the unconfining operations. Note that the Read Extractor extracts transaction read information from the SQL statements kept by the Transaction Proxy. The *Confinement Executor* is responsible for (1) maintaining the confinement time window as new malicious transactions are reported by the Intrusion Detector; (2) enforcing the damage confinement control with the help of the *U_SET*; and (3) maintaining the time stamp information by rewriting user SQL queries. Unconfining phases B and C are enforced by the *Unconfinement Executor*. Unconfining phases A and D are enforced by the *Repair Manager*, which also performs damage assessment and repair.

The key operations of the prototype are triggered by three main events. (1) When a new user transaction arrives, the Transaction Proxy will proxy the transaction, and the Unconfinement Executor will enforce the confinement control and maintain the time stamps for the data objects that are updated by this transaction. (2) When a new malicious transaction *B* is detected, the Confinement Executor will set a new confinement time window, the Unconfinement Executor will adjust the *U_SET* and its unconfining operations to cover *B*, and the Repair Manager will adjust its damage assessment and repair operations to cover *B*. (3) When the Repair Manager finishes the repair for the set of detected malicious transactions, the Unconfinement Executor will discontinue enforcing the confinement control.

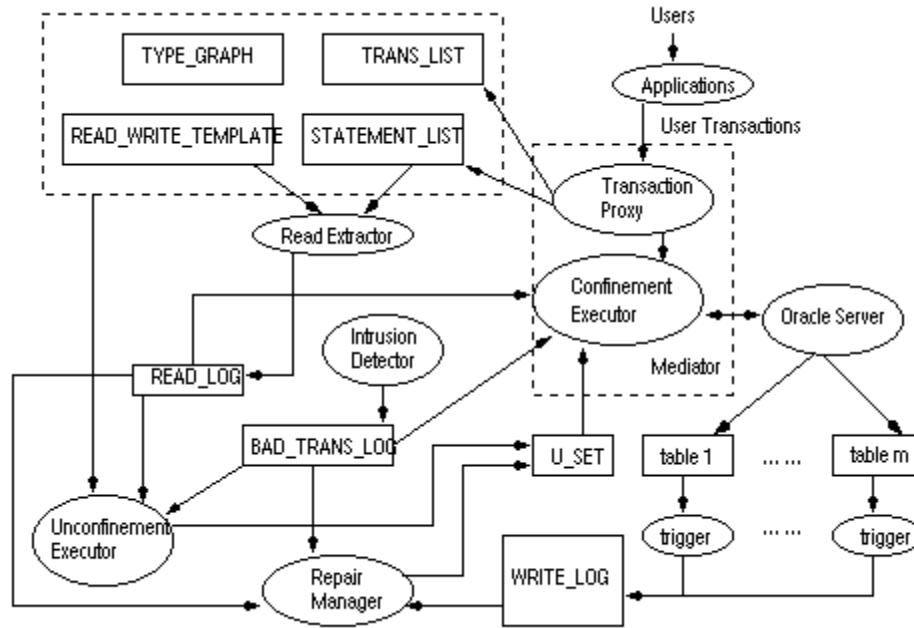


Figure 9 – Organization of the Damage Containment Subsystem Prototype

Architectures II and III share the same goal, that is, to reduce the extent of damage spreading, while they take two very different approaches. It should be noticed that these two architectures are actually complementary to each other and can be easily integrated into one, as illustrated in Figure 10.

6. Self-Stabilization Subsystem

The intrusion tolerance components introduced in Architectures I, II, and III can behave in many different ways. At one point of time, the *resilience* or *trustworthiness* of an intrusion tolerant database system is primarily affected by four factors: (a) the current attacks, (b) the current workload, (c) the current system state, and (d) the current defense *behavior* of the system. It is clear that based on the same system state, attack pattern, and workload, two intrusion tolerance database systems (of the same architecture) with different behaviors can yield very different levels of resilience. This suggests that one defense behavior is only good for a limited set of *environments*, which are determined by factors (a), (b), and (c). To achieve the maximum amount of resilience, intrusion tolerant systems must *adapt* their behaviors to the environment.

Architecture IV, as shown in Figure 10, integrates a *reconfiguration* framework to handle this challenge. In particular, an *Adaptor* is deployed to *monitor* the environment changes and *adjust* the behaviors of the intrusion tolerance components in a way such that the adjusted system behavior is more (cost) effective than the old system behavior in the changed environment.

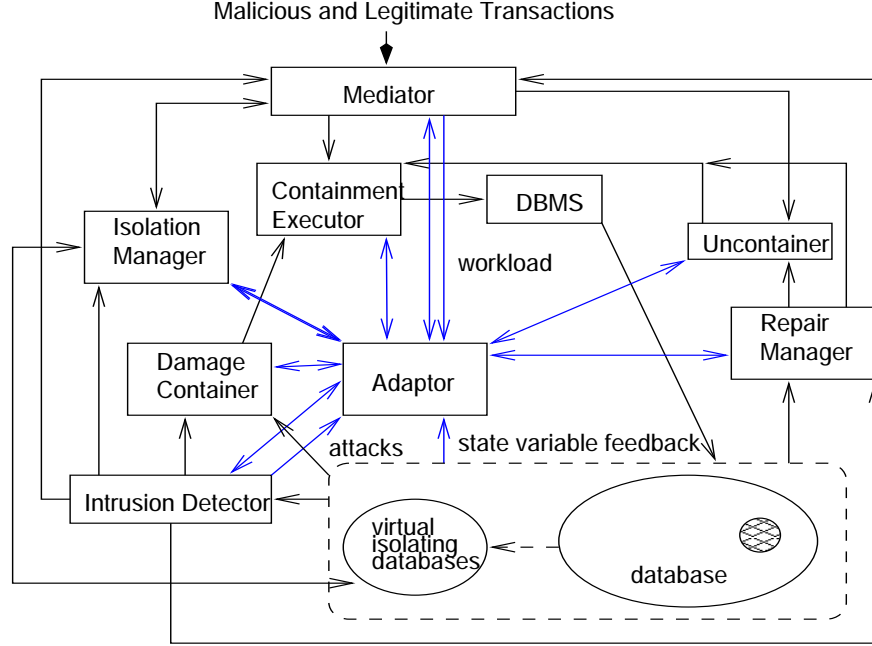


Figure 10 – Architecture IV

In Architectures I, II, and III, almost every intrusion tolerance component is reconfigurable and the *behavior* of each such component is *controlled* by a set of *parameters*. For example, the major control parameters for the Intrusion Detector are TH_m and TH_s . The major control parameter for the *Damage Container* is the amount of allowed damage *leakage*, denoted DL . When $DL = 0$, multi-phase containment is enforced; when there is no restriction on DL , one-phase containment is enforced. The major control parameter for the Mediator is the transaction delay time, denoted DT . When $DT = 0$, transactions are executed in full speed; when DT is not zero, transaction executions are slowed down. At time t , ITDB calls the set of control parameters (and the associated values) for an intrusion tolerance component C_i the *configuration* (vector) of C_i at time t , and the set of the configurations for all the intrusion tolerant components the configuration of the intrusion tolerant system at time t . In Architecture IV, each reconfiguration is done by adjusting the system from one configuration to another configuration.

The goal of Architecture IV is to improve the resilience of the system, which has three major aspects: (1) how well the level of data integrity is maintained in face of attacks; (2) how well the level of data and system availability is maintained in face of attacks; and (3) how well the level of cost effectiveness is maintained in face of attacks.

To do optimal reconfiguration, ITDB wants to find the best configuration (vector) for each (new) environment. However, this is very difficult, if not impossible, since the *adaptation space* of Architecture IV systems contains an exponential number of configurations. To illustrate, the simplest configuration of an Architecture IV system could be $[TH_m, TH_s, DL, DT]$, then the size of the adaptation space is $domain(TH_m) \times domain(TH_s) \times domain(DL) \times domain(DT)$, which is actually huge. Moreover, ITDB faces conflicting reconfiguration criteria, that is, trustworthiness and cost conflict with each other, and integrity and availability conflict with each other.

Therefore, ITDB envisions the problem of finding the best system configuration under multiple conflicting criteria a NP-hard problem.

Architecture IV focuses on near optimal heuristic adaptation algorithms which can have much less complexity. For example, a data integrity favored heuristic can work as follows: when the level of data integrity, i.e., LI , is below a specific warning threshold I_w , (a) switch the system to multi-phase containment, i.e., let $DL = 0$; (b) slow down the execution of new transactions by $DT = DT + \alpha(I_w - LI)$; and (c) lower the anomaly levels required for alarm raising, that is, $TH_m = TH_m - \beta(I_w - LI)$, and $TH_s = TH_s - \gamma(I_w - LI)$. In this way, ITDB rejects and isolates more transactions. Here the values of α , β , and γ are determined based on previous experiences. Note that it is very possible that different (value) combinations of (α, β, γ) are optimal for different environments. Hence it is worthy to have multiple such heuristics with different combinations of (α, β, γ) .

It is clear that under different environments different heuristics are the most effective. For example, in some cases integrity favored heuristics can be better, but in some other cases availability favored heuristics can be better. Architecture IV systems should have a mechanism to guide the system to pick the right heuristic (for the current environment). For example, a rule-based mechanism can be used for this purpose.

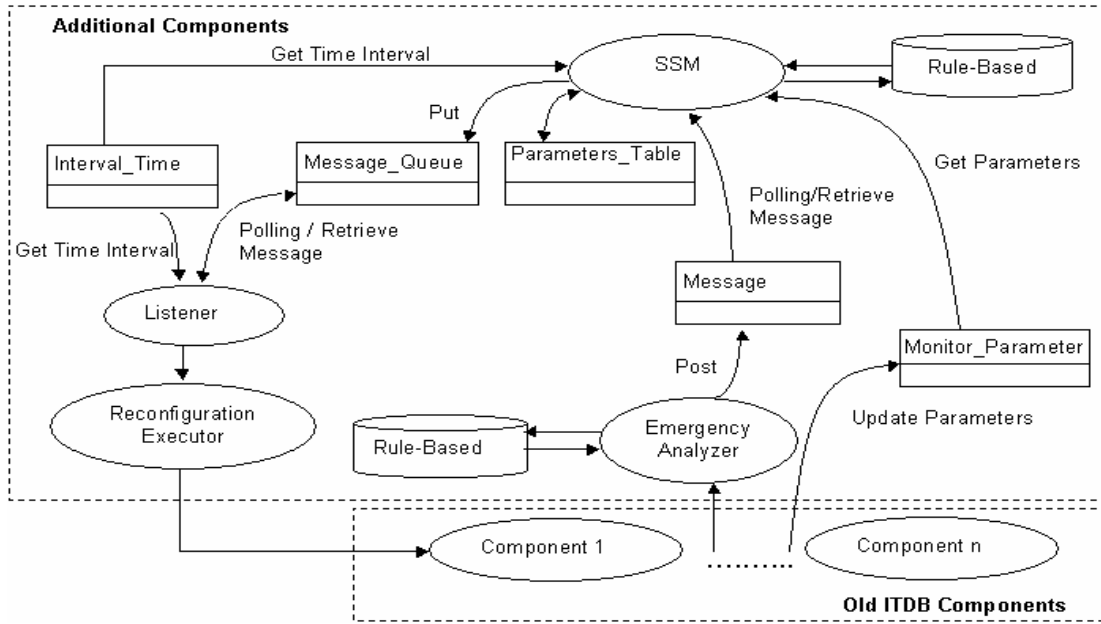


Figure 11 – Organization of the Self-Stabilization Subsystem Prototype

A rule-based self-stabilization subsystem prototype, which is shown in Figure 11, has been designed and implemented. In general, the adaptation strategies are programmed as *rules*. The

rules are fired by the *Self-Stabilization Manager* (SSM) and the corresponding reconfiguration operations are enforced by the *Reconfiguration Executor* through the *Listener*. The adaptations are triggered under three possible situations: (a) when the Emergency Analyzer reports an emergency to the SSM, (b) when the SSM pulls some situation data from the components, and (c) when the components push some situation data to the SSM. Readers can refer to [LL02] for more details about this subsystem.

7. ITDB Prototype

The prototypes for each ITDB subsystem have been integrated into the ITDB prototype. The ITDB prototype implements every functionality of the ITDB framework, including transaction proxying (the key function of the PEM), reads extraction, trail collection, intrusion detection, damage assessment, damage repair, multi-phase damage containment, attack isolation, and self-stabilization through dynamic reconfiguration. Moreover, two real world database applications have been implemented to test the functionality of ITDB: one for credit card transaction management, the other for inventory management (based on TPC-C).

The ITDB prototype has around 30,000 lines of (multi-threaded) C++ code and Oracle PL/SQL code. Each component of ITDB is implemented as a set of C++ objects that have a couple of CORBA calling interfaces through which other components can interact with the component and the reconfiguration can be done. ITDB uses ORBacus V4.0.3 as the ORB. Finally, ITDB assumes that applications use OCI calls, a standard interface for Oracle, to access the database, and ITDB proxies transactions at the OCI call level. The reason that ITDB does not proxy transactions at the TCP/IP or the SQL*NET level, which is more general, is because the exact packet structure of SQL*NET is confidential.

One possible installation of the ITDB prototype is shown in Figure 12. Enabled by the ORB-based system design, ITDB can distribute its components across a network in a variety of ways for load-balancing and improved resilience. In addition to evaluating each subsystem prototype, the integrated ITDB prototype has been evaluated from the functionality perspective. The results show that the designed functionalities of ITDB are achieved and the ITDB components collaborate with each other in a smooth way.

8. ITDB Validation Framework

A validation framework for ITDB is developed under the guideline of the OASIS program. The validation framework clearly identifies the assumptions made by this project, justifies the accomplishments of this project, and identifies the limitations of the ITDB framework. Readers can refer to [LV02] for more details about this validation framework.

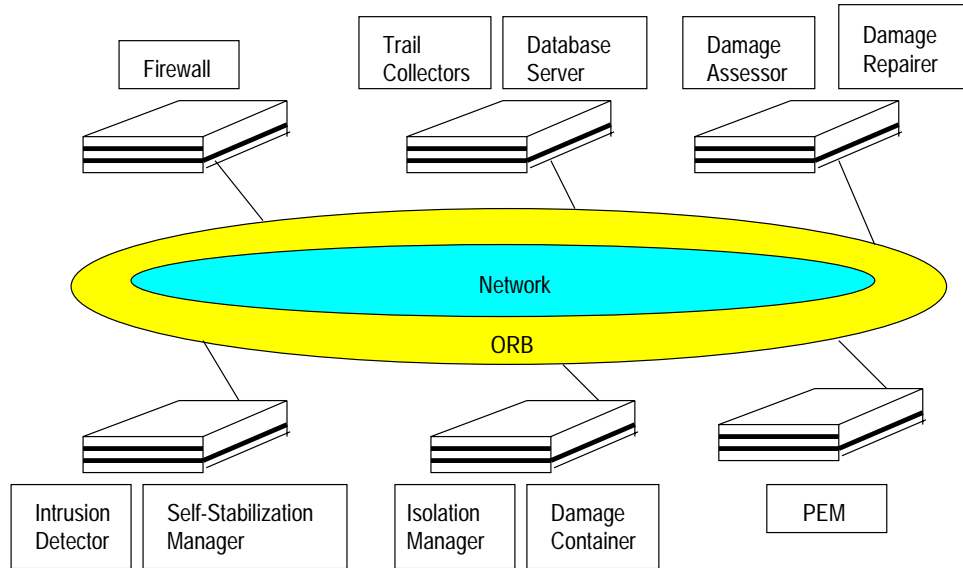


Figure 12 – ITDB Prototype Installation

9. Attack Prediction

A drawback of existing secure system designs is that they focus on the system itself and consider *attackers* as a part of the *environment* of the system. As a result, such designs can only passively observe and react to the environment, especially the attacks, but cannot model, analyze, and predict the attacker intent, objective, and strategies.

To enable attack prediction, existing secure system designs are extended into a new paradigm, which is shown in Figure 13, where the attacks are no longer a part of the environment. In particular, the attacker and the system are modeled as two *peer* systems, or two *players* fighting a sequence of *battles* or *game plays*, where (a) each player has a set of *strategies* to fight. A strategy can be an *action* or a sequence of actions. (b) The *strategy space* of the system is determined by the set of security facilities (or components) deployed to protect the system (Note that for clarity these components are not shown in Figure 13). The system can defend against the attacker in many different manners by having multiple ways to configure its facilities. Each such manner can be a defense strategy. (c) The *strategy space* of the attacker is the set of attacks that the attacker is able to launch. An attack can be an action or a sequence of actions. (d) At one point of time, the battle is defined by a pair of strategies: one from the attacker, one from the system. (e) The *outcome* of each battle indicates “who wins” in this round. In the real world, an outcome could be “the attacker breaks in”, “a malicious access request is rejected”, etc. Note that for some battles, there may not be clear winners. (f) A battle-outcome yields two *utility* measures: one earned by the attacker, the other earned by the system. These utility measures indicate how the two players *prefer* the outcome. The framework uses utility measures to precisely define the meaning of “winning a game”. (g) The goal of each player is to win the game, or to maximize his or her utilities. (h) The *environment* now only contains the good accesses. (i) Each player maintains a *knowledge base* to keep the player's knowledge about the other player and the other player's belief. (j) Each player selects the strategy to play based on his or her knowledge base. (k) Before each player fights a new battle, the outcomes of previous battles are already known, and become a part of each player's knowledge base. (l) The attacker's

uncertainty about the system's defense, and the system's *uncertainty* about the attacker's offense, are all modeled by the rationality notion of an *expected-utility maximizer*. (m) The system's *uncertainty* about “whether or not the incoming access is an attack” is modeled by having multiple *types* of players that play with the system.

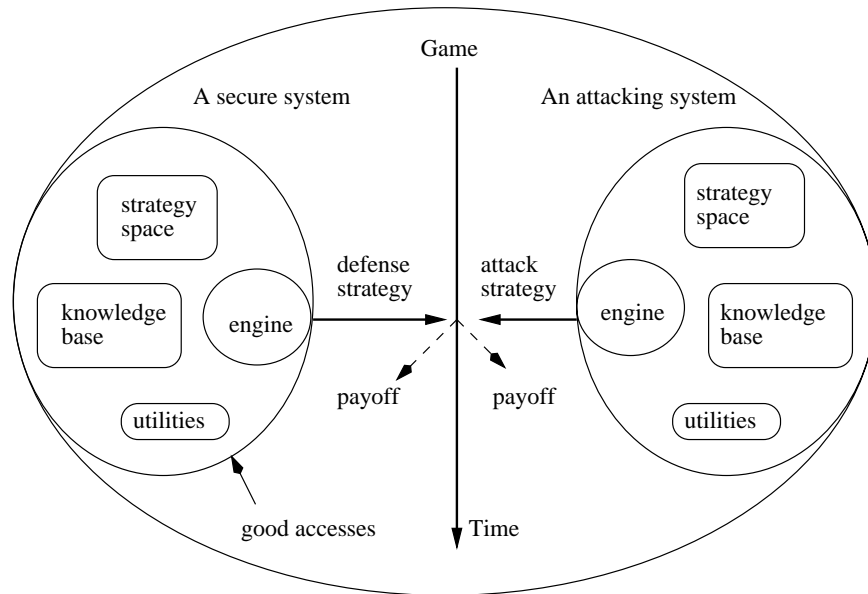


Figure 13 – Model of Game Theoretic Attack Prediction

A direct *output* of the above game theoretic attacker-system model is *predictions* about attacker actions (and strategies). In particular, the above attacker-system model is mathematically formalized as a multi-stage Bayesian game, and it is found that the *Nash equilibrium* of such games can produce valuable attack predictions since (1) the game model captures the key components of real world attacker-system relationships, such as strategies, outcomes, utilities (or incentives), knowledge and uncertainty; (2) the game model captures such key characteristics of real world attacker-system relationships as *incentive-based* strategy selection, strategic independence, and knowledge-based strategy selection; and (3) the notion of Nash equilibrium captures such key characteristics of real world attacker-system relationships as *relativity* in (best) strategy selection, and the *rationality* notion of an expected-utility maximizer.

A general game theoretic attack prediction model is developed to predict the attacks on IDS-protected systems, and a concrete game theoretic attack prediction model is developed to predict credit card fraud transactions. Within this project, extensive simulations have been done on the game plays involved in the credit card fraud prediction model, and the results show that (a) there exists pure strategy Nash equilibrium, (b) the Nash equilibria indicate the best strategies for *rational* attackers, and (c) game theoretic attack prediction is typically computation intensive, and *approximation* is usually needed for practical game theoretic attack prediction. Readers can refer to [LLI02] for more details on this accomplishment.

10. References

- [AJL02] P. Ammann, S. Jajodia, and P. Liu. Recovering from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2002, To appear.
- [BGI00] D. Barbara, R. Goel, and S. Jajodia. Using Checksums to Detect Data Corruption. In *Proc. 2000 International Conference on Extending Data Base Technology*, Mar 2000.
- [D84] S. B. Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Database Systems*, 9(3):456-481, 1994.
- [LIM01] P. Liu, S. Ingsriswang, and X. Ma. AAID: An Application Aware Transaction-Level Database Intrusion Detection System. Technical Report, University of Maryland at Baltimore County, Dec 2001.
- [Liu00] P. Liu. The General Design of ItDBMS. Technical Report, University of Maryland at Baltimore County, July 2000.
- [Liu01] P. Liu. DAIS: A Real-Time Data Attack Isolation System for Commercial Database Applications. *Proc. 2001 Annual Computer Security Applications Conference*, pages 219-229. IEEE Computer Press.
- [Liu02] P. Liu. Architectures for Intrusion Tolerant Database Systems. *Proc. 2002 Annual Computer Security Applications Conference*, To appear.
- [LJ01] P. Liu and S. Jajodia. Multi-Phase Damage Confinement in Database Systems for Intrusion Tolerance. *Proc. 14th IEEE Computer Security Foundations Workshop*, June 2001, pages 191-205.
- [LJB01] P. Liu and S. Jajodia. *Trusted Recovery and Defensive Information Warfare*. Kluwer Academic Publishers, 2001.
- [LJAL02] P. Liu, S. Jajodia, P. Ammann, and J. Li. Can-Follow Concurrency Control. *Proc. 2002 IASTED International Conference on Networks, Parallel and Distributed Processing and Applications*, To appear.
- [LJLI01] P. Liu, J. Jing, P. Luenam, and S. Ingsriswang. Intrusion Tolerant Database Systems. Technical Report, University of Maryland at Baltimore County, April 2001.
- [LJM99] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion Confinement by Isolation in Information Systems. *Journal of Computer Security*, 8(4):243-279, 2000.

- [LLWJ02] P. Liu, P. Luenam, Y. Wang, J. Jing, and S. Ingsriswang. ITDB: An Intrusion Tolerant Database System. Demo Paper. Technical Report, University of Maryland at Baltimore County, Feb 2002.
- [LL01] P. Luenam and P. Liu. ODAR: An On-the-fly Damage Assessment and Repair System for Commercial Database Applications. *Proc. 15th IFIP WG11.3 Working Conference on Data and Application Security*, July 2001.
- [LL02] P. Luenam and P. Liu. The Design of an Adaptive Intrusion Tolerant Database System. *Proc. 2002 IEEE Workshop on Intrusion Tolerant Database Systems*, June 2002.
- [LLI02] P. Liu and L. Li. A Game Theoretic Approach to Attack Prediction. Technical Report, University of Maryland at Baltimore County, April 2002.
- [Lunt93] T. F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405-418, June 1993.
- [LV02] Peng Liu. ITDB Survivability Validation Framework. Technical Report, Pennsylvania State University, August 2002.
- [LW02] P. Liu and Y. Wang. The Design and Implementation of a Multiphase Damage Confinement System. *Proc. 16th IFIP WG11.3 Working Conference on Data and Application Security*, July 2002.
- [LX01] P. Liu and H. Xu. Efficient Damage Assessment and Repair in Resilient Distributed Database Systems. *Proc. 15th IFIP WG11.3 Working Conference on Data and Application Security*, July 2001.
- [MG96] J. McDermott and D. Goldschlag. Towards a Model of Storage Jamming. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 176-185, Ireland, June 1996.
- [MHL94] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network Intrusion Detection. *IEEE Network*, June 1994, pages 26-41.
- [MVS00] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proc. of 4th Symposium on Operating System Design and Implementation*, San Diego, CA, Oct 2000.
- [Tylor00] C. Taylor. Behind the Hack Attack. *Time*, (2): 45-47, February 2000.